

4.2 Assembly Language

Name: _____ Class: _____ Date: _____ Total: 43 marks

- KEY WORDS** assembly language • assembler • machine code
- addressing mode • index register • relative addressing
 - symbol table • instruction group

Objective

Build the skills to answer exam questions on **assembly language** : how it relates to machine code, the two-pass assembler, the five modes of addressing, the instruction groups, and tracing a program.

You must be able to:

- explain the relationship between **assembly language** and **machine code**
- describe the two passes of a **two-pass assembler**
- use the five **modes of addressing** : immediate, direct, indirect, indexed, relative
- name the **group** of an instruction (data movement, input/output, arithmetic, jump, compare)
- give the contents of the ACC after each instruction, and **trace** a program in a trace table

The past-paper set for this sheet also asks about the registers PC, MAR, MDR and CIR, and about clock speed and cache. Revise those on sheet 4.1.

1 Key knowledge and worked examples

Study these first. Each block gives a fact you must know or the method for one question type.

■ Key knowledge: assembly language and the assembler

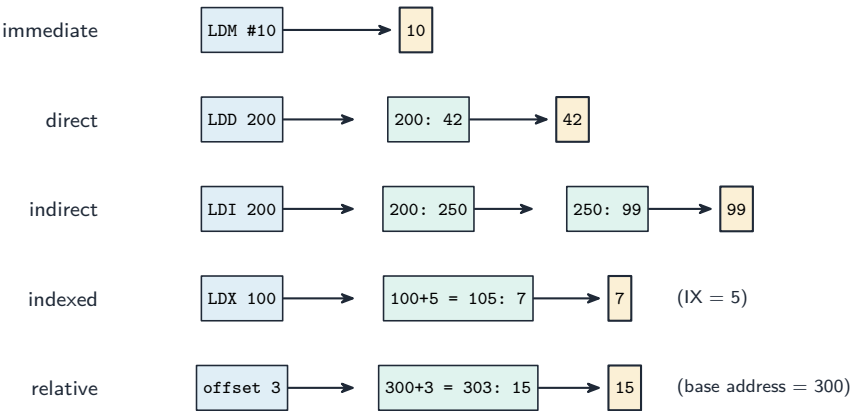
Term	Meaning
machine code	binary instructions the CPU runs directly
assembly language	the same instructions written as mnemonics
one-to-one assembler	one assembly instruction = one machine-code instruction
assembler	translates assembly language into machine code

A **two-pass assembler** reads the program twice:

- **Pass 1:** read each line, and put every **label** and its address into the **symbol table**. No code is made yet.
- **Pass 2:** turn each instruction into machine code. For a label, look up its address in the symbol table.
- **Why two passes?** A jump can name a label that appears **later** in the program (a **forward reference**). After pass 1, every label's address is known.

■ Key knowledge: the five modes of addressing

The mode says **how the CPU finds the value** to use. In the figure, 200: 42 means "address 200 holds 42".



Assembly instructions use addressing modes to locate their operands

Mode	The value used is...	Example
immediate	the operand itself	LDM #10 → 10
direct	at the address given	LDD 200 → what 200 holds
indirect	at the address stored there	LDI 200 → what 250 holds
indexed	at address + IX	LDX 100 → what 105 holds
relative	at base address + offset	offset 3 → what 303 holds

- **Indirect:** address 200 does not hold the value. It holds **another address** (250), and the value is there.
- LDR #n puts the number *n* into the **index register** IX.
- **Relative** (mark-scheme words): the operand is an **offset**, added to a **base** address to give the address used.

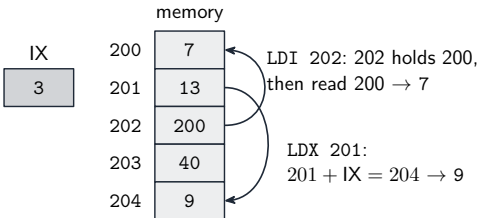
■ Key knowledge: instruction groups

Group	Examples
data movement (copy a value)	LDM LDD LDI LDX LDR STO
input and output	IN OUT
arithmetic (change a number)	ADD SUB INC DEC
unconditional jump (always jump)	JMP
conditional jump (jump only if...)	JPE JPN
compare	CMP

- INC IX is **arithmetic**: it adds 1 to the index register.
- CMP does **not** change the ACC. It only records whether the two values were equal.

■ Worked example: what is in the ACC after each instruction?

Memory holds 200: 7, 201: 13, 202: 200, 203: 40, 204: 9, and IX = 3. Each instruction runs on its own.



Instruction	Mode	Address read	ACC
LDM #202	immediate	none	202
LDD 202	direct	202	200
LDI 202	indirect	202 holds 200, so read 200	7
LDX 201	indexed	201 + 3 = 204	9

- **Step 1**: look at the operand. # means the number itself; no # means an address.
- **Step 2**: work out the address the CPU reads. **Step 3**: write what that address holds.
- Trap: LDI reads memory **twice**. 202 is only a pointer to the address 200.

■ Worked example: bit patterns in the ACC

The exam also writes the ACC as 8 bits. & means hexadecimal: each hex digit is 4 bits (&F0 = 1111 0000). Memory holds 150: 1010 1010.

ACC before	Instruction	ACC after	Why
0000 1111	AND 150	0000 1010	1 only where both bits are 1
any value	LDM #20	0001 0100	20 in 8-bit binary
0011 0011	XOR &F0	1100 0011	1 where the bits differ
0110 0000	CMP 150	0110 0000	CMP leaves the ACC unchanged

Sheet 4.3 practises AND, OR, XOR and the shifts in detail.

■ Worked example: tracing a program with a loop

Memory location 100 holds 0. JPN jumps if the last compare was **not equal**.

```

10 LDM #0
11 STO 100
12 LDD 100
13 ADD #2
14 STO 100
15 CMP #4
16 JPN 12
17 OUT
18 END

```

Write one row for each instruction **executed**, in the order it runs. Write a value in a column only when that instruction **puts a value there** (a load, a calculation or a store).

Address	ACC	100	Note
10	0		
11		0	
12	0		
13	2		
14		2	
15, 16			2 ≠ 4, so jump to 12
12	2		
13	4		
14		4	
15, 16			4 = 4, so no jump
17			output 4

- After a jump, the next row is the jump's **target**. Lines that are jumped over get no row.

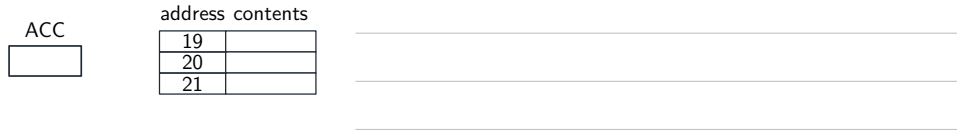
2 Practice

Now apply the methods above.

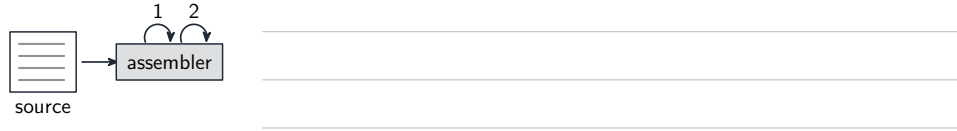
2.1 State the difference between **assembly language** and **machine code**. [2]



2.2 State what each instruction loads into the ACC: (a) LDM #20 (b) LDD 20. [2]



2.3 Explain why an assembler needs **two passes**. [2]



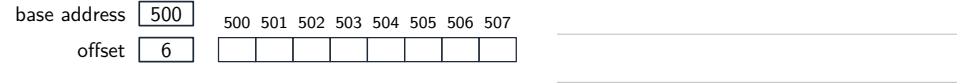
2.4 LDX 100 is executed when the index register IX = 4. State which address is read. [1]



2.5 Name the instruction group of each instruction: (a) LDD 50 (b) INC IX (c) JPE 30 (d) CMP #5 (e) OUT. [5]



2.6 (a) State what is meant by **relative addressing**. [1]



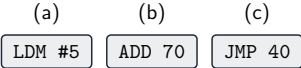
(b) The base address is 500 and the offset is 6. State the address that is read. [1]

3 Exam-style questions

3.1 Give one reason a programmer might write part of a program in assembly language rather than a high-level language. [1]



3.2 Assembly language instructions are grouped. Complete each statement with the name of the group. [3]



- (a) LDM #5 is an example of an instruction in the group.
(b) ADD 70 is an example of an instruction in the group.
(c) JMP 40 is an example of an instruction in the group.

3.3 Trace this program (start: a = 10, b = 4) and complete the table. [3]

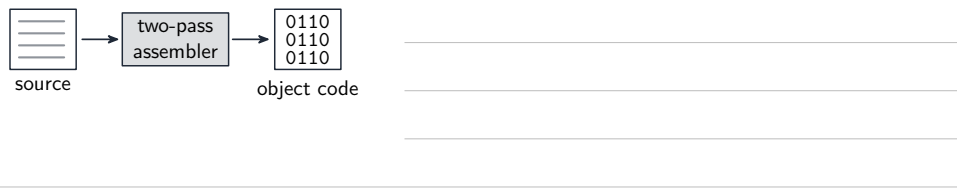


3.4 (a) Explain the difference between **immediate**, **direct** and **indirect** addressing. [3]



(b) State what is meant by **indexed** addressing. [1]

3.5 (a) Describe what happens in **pass 1** and in **pass 2** of a two-pass assembler. [4]



(b) State why two passes are needed. [1]

3.6 The figure shows the contents of memory addresses 60 to 64. The ACC holds 0 and IX holds 0 before each program runs. Give the contents of the ACC after each program has run. [4]

ACC

IX

address	contents
60	64
61	9
62	60
63	20
64	35

Program	Instructions	ACC after the program
1	LDM #62	
2	LDD 62, then ADD 61	
3	LDI 62	
4	LDR #2, then LDX 61	

3.7 Memory holds 200: 1100 1010 and 201: 0011 0110. Each row starts from the ACC value shown. Give the contents of the ACC after each instruction. [4]

ACC

address	contents
200	1100 1010
201	0011 0110

ACC before	Instruction	ACC after
1111 0000	AND 201	
0000 0000	LDM #75	
0000 0110	XOR &0F	
0101 0101	CMP 200	

3.8 Memory holds 70: 4, 71: 6, 72: 70 and 73: 71. Trace the program by completing the trace table. JPE jumps if the last compare was **equal**. [5]

150 LDD 71
151 DEC ACC
152 STO 73
153 LDI 72
154 CMP 73
155 JPE 159
156 LDD 73
157 ADD #3
158 JMP 160
159 INC ACC
160 STO 70
161 END

ACC

address	contents
70	4
71	6
72	70
73	71

Instruction address	ACC	70	71	72	73
		4	6	70	71

Solutions

Each answer shows the steps, then the **result**.

2.1

- machine code is **binary** that the CPU runs directly
- assembly language is a **readable** version using mnemonics, with one assembly instruction for each machine-code instruction

2.2

- (a) the value **20** (immediate)
- (b) the **contents of address 20** (direct)

2.3

- a jump can refer to a label defined **later** (a forward reference)
- pass 1 records every label's address first, so pass 2 can fill it in

2.4

- indexed: base 100 + IX 4 $\rightarrow$ address **104**

2.5

- (a) **data movement**
- (b) **arithmetic**
- (c) **conditional jump**
- (d) **compare**
- (e) **input and output**

2.6

- (a) the operand is an **offset**, which is added to a **base address** to give the address used
- (b) $500 + 6 \rightarrow$ address **506**

3.1

- e.g. to control hardware directly / for speed / for a small, tight routine

3.2

- (a) **data movement**
- (b) **arithmetic** (operations)
- (c) **unconditional jump**

3.3

- start: $a = 10$, $b = 4$

Instruction	ACC	c
LDD a	10	-
ADD b	14	-
STO c	14	14
END	14	14

- ACC after ADD = **14**
- c after STO = **14**

3.4

(a)

- immediate: the operand **is the value** to use
- direct: the operand is the **address** of the value
- indirect: the operand is an address that holds **another address**, which holds the value

(b) the address used is the operand **plus the contents of the index register** (IX)

3.5

(a) **pass 1**: read the source and build the symbol table: record the address of each label

- **pass 2**: translate each instruction to machine code, looking up label addresses in the symbol table

(b) to resolve **forward references** (a jump to a label defined later)

3.6

- program 1: immediate, so the number itself: **62**
- program 2: 62 holds 60, then add what 61 holds: $60 + 9 =$ **69**
- program 3: 62 holds 60, so read address 60: **64**
- program 4: IX = 2, address $61 + 2 = 63$, which holds **20**

3.7

- AND 201: $1111\ 0000$ AND $0011\ 0110 =$ **0011 0000**
- LDM #75: $75 = 64 + 8 + 2 + 1 =$ **0100 1011**
- XOR &0F: $\&0F = 0000\ 1111$, so the low four bits flip: **0000 1001**
- CMP 200: compare only, so ACC is unchanged: **0101 0101**

3.8

Address	ACC	70	71	72	73
		4	6	70	71
150	6				
151	5				
152					5
153	4				
154					
155					
156	5				
157	8				
158					
160		8			
161					

- 150–152: $ACC = 6$, then 5, stored in 73
- 153: 72 holds 70, and 70 holds 4, so $ACC = 4$
- 154–155: $4 \neq 5$, so no jump to 159
- 156–158: $ACC = 5$, then $5 + 3 = 8$; **JMP 160** skips 159
- 160: 70 becomes **8**; the program ends at 161